

## APPENDIX A

```
namespace System.Storage
```

```
{
    abstract class ItemContext : IDisposable, IServiceProvider
    {
```

**ItemContext Creation and Management Members**

```
// Applications cannot create ItemContext objects directly nor can they derive
// classes from ItemContext.
internal ItemContext();
```

```
// Create ItemContext that can be used to search the specified paths or, if no path
// is specified, the default store on the local computer.
public static ItemContext Open();
public static ItemContext Open( string path );
public static ItemContext Open( params string[] paths );
```

```
// Return the paths specified when the ItemContext was created.
public string[] GetOpenPaths();
```

```
// Create a copy of this ItemContext. The copy will have independent transaction, caching
// and update state. The cache will initially be empty. It is expected that using a
// cloned ItemContext would be more efficient than opening a new ItemContext using the
// same item domain(s).
public ItemContext Clone();
```

```
// Close the ItemContext. Any attempt to use the ItemContext after it is closed will
// result in an ObjectDisposedException.
public void Close();
void IDisposable.Dispose();
```

```
// True if any domain specified when the ItemContext was opened resolved to a remote
// computer.
public bool IsRemote { get; }
```

```
// Returns an object that can provide the requested service type. Returns null if the
// requested service cannot be provided. The use of the IServiceProvider pattern allows
// API that are not normally used and could confuse developers to be factored out of
// the ItemContext class. ItemContext can provide the following kinds of services:
// ItemSerialization, IStoreObjectCache
public object GetService( Type serviceType );
```

**Update Related Members**

```
// Saves changes represented by all modified objects and all objects passed to
// MarkForCreate or MarkForDelete. May throw UpdateCollisionException if an update
// collision is detected.
public void Update();
```

```
// Saves changes represented by the specified objects. The objects must have either
// been modified or passed to MarkForCreate or MarkForDelete, otherwise Argument-
// Exception is thrown. May throw UpdateCollisionException if an update collision is
```

```

// detected.
public void Update( obj ct obj ctToUpdate );
public void Update( IEnumerable objectsToUpdate );

// Refreshes the content of the specified objects from the store. If the object has
// been modified, the changes are overwritten and the object is no longer considered
// modified. Throws ArgumentException if anything other than an item, item extension,
// or relationship object is specified.
public void Refresh( object objectToRefresh );
public void Refresh( IEnumerable objectsToRefresh );

// Raised when an update detects that data has been changed in the store between when a
// modified object was retrieved and an attempt was made to save it. If no event handler
// is registered, the update throws an exception. If an event handler is registered, it
// can throw an exception to abort the update, case the modified object to overwrite
// the data in the store or merge the changes made in the store and in the object.
public event ChangeCollisionEventHandler UpdateCollision;

// Raised at various points during update processing to provide update progress
// information.
public event UpdateProgressEventHandler UpdateProgress;

// Async versions of Update
public IAsyncResult BeginUpdate( IAsyncCallback callback, object state );
public IAsyncResult BeginUpdate( object objectToUpdate,
                                IAsyncCallback callback,
                                object state );
public IAsyncResult BeginUpdate( IEnumerable objectsToUpdate,
                                IAsyncCallback callback,
                                object state );
public void EndUpdate( IAsyncResult result );

// Async versions of Refresh
public IAsyncResult BeginRefresh( object objectToRefresh,
                                IAsyncCallback callback,
                                object state );
public IAsyncResult BeginRefresh( IEnumerable objectsToRefresh,
                                IAsyncCallback callback,
                                object state );
public void EndRefresh( IAsyncResult result );

```

#### **Transaction Related Members**

```

// Begins a transaction with the specified isolation level. The default isolation level
// is ReadCommitted. In all cases, a distributed transaction is started because it may
// have to encompass changes stream typed item properties.
public Transaction BeginTransaction();
public Transaction BeginTransaction( System.Data.IsolationLevel isolationLevel );

```

#### **Search Related Members**

```

// Create an ItemSearcher that will search this item context for objects of the
// specified type. Throws ArgumentException if a type other than an item,
// relationship, or item extension is specified.

```

```

public ItemSearcher GetSearcher( Type type );

// Find an item given its id.
public Item FindItemById( ItemId itemId );

// Find an item given its path. The path may be absolute or relative. If it is relative,
// NotSupportedException will be thrown if multiple item domains were specified when
// the ItemContext was opened. Will return null if no such item exists. Creates a
// connection to the \machine\share part of the domain to retrieve the item. The
// item will be associated with that domain.
public Item FindItemByPath( string path );

// Find an item given its path. The path is relative to the specified item domain.
// Creates a connection to the specified domain to retrieve the item. The item will be
// associated with that domain. Will return null if no such item exists.
public Item FindItemByPath( string domain, string path );

// Find a set of items given a path. The path is relative to the item domains specified
// when the ItemContext was opened. Will return an empty result if no such item exists.
public FindResult FindAllItemsByPath( string path );

// Find a relationship given its ids.
public Relationship FindRelationshipById( ItemId itemId,
                                     RelationshipId relationshipId );

// Find a item extension given its ids.
public ItemExtension FindItemExtensionById( ItemId itemId,
                                           ItemExtensionId itemExtensionId );

// Find all item, relationship, or item extensions of the specified type optionally
// satisfying a given filter. Throws ArgumentException if a type other than one of
// these is specified.
public FindResult FindAll( Type type );
public FindResult FindAll( Type type, string filter );

// Find any item, relationship, or item extensions of the specified type that satisfies
// a given filter. Throws ArgumentException if a type other than one of these is
// specified. Returns null if no such object is found.
public object FindOne( Type type, string filter );

// Find the item, relationship, or item extensions of the specified type that satisfies
// a given filter. Throws ArgumentException if a type other than one of these is
// specified. Throws ObjectNotFoundException if no such object was found. Throws
// MultipleObjectsFoundException if more than one object was found.
public object FindOnly( Type type, string filter );

// Returns true if an item, relationship, or item extensions of the specified type that
// satisfies a given filter exists. Throws ArgumentException if a type other than one
// of these is specified.
public bool Exists( Type type, string filter );

// Specifies how the objects returned by a search relate to the object identity map
// maintained by the ItemContext.

```

```

public SearchCollisionMode SearchCollisionMode { get; set; }

// Raised when PreserveModifiedObjects is specified for ResultMapping. This event allows
// the application to selectively update the modified object with data retrieved with the
// search.
public event ChangeCollisionEventHandler SearchCollision;

// Incorporate an object from another ItemContext into this item context. If an object
// representing the same item, relationship or item extension does not already exist
// in this ItemContext's identity map, a clone of the object is created and added to
// the map. If an object does exist, it is updated with the state and content of the
// specified object in a way consistent with the SearchCollisionMode.
public Item IncorporateItem( Item item );
public Relationship IncorporateRelationship( Relationship relationship );
public ItemExtension IncorporateItemExtension( ItemExtension itemExtension );

}

// Handler for ItemContext.UpdateCollision and ItemSearcher.SearchCollision events.
public delegate void ChangeCollisionEventHandler( object source,
        ChangeCollisionEventArgs args );

// Arguments for the ChangeCollisionEventHandler delegate.
public class ChangeCollisionEventArgs : EventArgs
{
    // Modified item, item extension, or relationship object.
    public object ModifiedObject { get; }

    // Properties from store.
    public IDictionary StoredProperties { get; }
}

// Handler for ItemContext.UpdateProgress.
public delegate void UpdateProgressEventHandler( ItemContext itemContext,
        UpdateProgressEventArgs args );

// Arguments for the UpdateProgressEventHandler delegate
public class ChangeCollisionEventArgs : EventArgs
{
    // The current update operation.
    public UpdateOperation CurrentOperation { get; }

    // The object that is currently being updated.
    public object CurrentObject { get; }
}

// Specifies how the objects returned by a search relate to the objects identity map
// maintained by the ItemContext.
public enum SearchCollisionMode
{
    // Indicates that new objects should be created and returned. Objects representing the
    // same item, item extension, or relationship in the identity map are ignored. If this
    // option is specified the SearchCollision event will not be raised.
    DoNotMapSearchResults,

```

```

// Indicates that objects from the identity map should be returned. If the content of
// an object has been modified by the application, the modified object's content is
// preserved. If the object has not been modified, its content is updated with the
// data returned by the search. The Application may provide an handler for the
// SearchCollision event and selectively update the object as desired.
PreserveModifiedObjects,

// Indicates that the objects from the identity map should be returned. The content
// of the object is updated with the data returned by the search, even if the object
// has been modified by the application. If this option is specified the Search-
// Collision event will not be raised.
OverwriteModifiedObjects
}

// The current update operation.
public enum UpdateOperation
{
    // Provided when Update is first called. CurrentObject will be null.
    OverallUpdateStarting,

    // Provided just before Update returns after a successful update. CurrentObject will be
    // null.
    OverallUpdateCompletedSucessfully,

    // Provided just before Update throws an exception. CurrentObject will be the exception
    // object.
    OverallUpdateCompletedUnsuccessfully,

    // Provided when the update of an object is started. CurrentObject will reference the
    // object that will be used for the updated.
    ObjectUpdateStaring,

    // Provided when a new connection is needed. CurrentObject will be a string that contains
    // the path identifying an item domain as passed to ItemContext.Open or retrieved from
    // the Location field of a relationship.
    OpeningConnection
}
}

```

*[Remainder of Page Intentionally Left Blank]*

## APPENDIX B

```
namespace Syst m.Storage
{
```

```
// Executes a search across a specific type in an item context.
```

```
public class ItemSearcher
```

```
{
```

**Constructors**

```
public ItemSearcher();
```

```
public ItemSearcher( Type targetType, ItemContext context );
```

```
public ItemSearcher( Type targetType, ItemContext context,  
    params SearchExpression[] filters );
```

**Properties**

```
// The filters used to identify matching objects.
```

```
public SearchExpressionCollection Filters {get;}
```

```
// The ItemContext that specifies the domains that will be searched.
```

```
public ItemContext ItemContext {get; set;}
```

```
// The search parameter collection.
```

```
public ParameterCollection Parameters {get;}
```

```
// The type the searcher will operate against. For simple searches this is the type of  
// the object that will be returned.
```

```
public Type TargetType {get; set;}
```

**Search Methods**

```
// Find objects of TargetType that satisfy the conditions specified by Filters. Returns
```

```
// an empty FindResult if no such objects exist.
```

```
public FindResult FindAll();
```

```
public FindResult FindAll( FindOptions findOptions );
```

```
public FindResult FindAll( params SortOption[] sortOptions );
```

```
// Find any one object of TargetType that satisfies the conditions specified by Filters.
```

```
// Returns null if no such object exists.
```

```
public object FindOne();
```

```
public object FindOne( FindOptions findOptions );
```

```
public object FindOne( params SortOption[] sortOptions );
```

```
// Find the object of TargetType that satisfies the conditions specified by Filters.
```

```
// Throws ObjectNotFoundException if no such object was found. Throws MultipleObjects-
```

```
// FoundException if more than one object was found.
```

```
public object FindOnly();
```

```
public object FindOnly( FindOptions findOptions );
```

```
// Determine if an object of TargetType that satisfies the conditions specified by
```

```
// Filters exists.
```

```
public bool Exists();
```

```

// Creates an object that can be used to more efficiently execute the same search
// repeatedly.
public PreparedFind Prepare Find();
public PreparedFind PrepareFind( FindOptions findOptions );
public PreparedFind PrepareFind( params SortOption[] sortOptions );

// Retrieves the number of records that would be returned by FindAll().
public int GetCount();

// Asynchronous versions of various methods.
public IAsyncResult BeginFindAll( AsyncCallback callback,
    object state );

public IAsyncResult BeginFindAll( FindOptions findOptions,
    AsyncCallback callback,
    object state );

public IAsyncResult BeginFindAll( SortOption[] sortOptions,
    AsyncCallback callback,
    object state );

public FindResult EndFindAll( IAsyncResult ar );

public IAsyncResult BeginFindOne( AsyncCallback callback,
    object state );

public IAsyncResult BeginFindOne( FindOptions findOptions,
    AsyncCallback callback,
    object state );

public IAsyncResult BeginFindOne( SortOption[] sortOptions,
    AsyncCallback callback,
    object state );

public object EndFindOne( IAsyncResult asyncResult );

public IAsyncResult BeginFindOnly( AsyncCallback callback,
    object state );

public IAsyncResult BeginFindOnly( FindOptions findOptions,
    AsyncCallback callback,
    object state );

public IAsyncResult BeginFindOnly( SortOption[] sortOptions,
    AsyncCallback callback,
    object state );

public object EndFindOnly( IAsyncResult asyncResult );

public IAsyncResult BeginGetCount( AsyncCallback callback,
    object state );

public int EndGetCount( IAsyncResult asyncResult );

```

```

public IAsyncResult B ginExists( AsyncCallback callback,
    object stat );

public bool EndExists( IAsyncResult asyncResult );

// Options used when executing a search.
public class FindOptions
{

    public FindOptions();

    public FindOptions( params SortOption[] sortOptions );

    // Specifies if delay loadable fields should be delay loaded.
    public bool DelayLoad {get; set;}

    // The number of matches that are returned.
    public int MaxResults {get; set;}

    // A collection of sort options.
    public SortOptionCollection SortOptions {get;}
}

// Represents a parameter name and value.
public class Parameter
{
    // Initializes a Parameter object with a name and value.
    public Parameter( string name, object value );

    // The parameter's name.
    public string Name {get;}

    // The parameter's value.
    public object Value {get; set;}
}

// A collection of parameter name/value pairs.
public class ParameterCollection : ICollection
{

    public ParameterCollection();

    public int Count {get;}

    public object this[string name] {get; set;}

    public object SyncRoot {get;}

    public void Add( Parameter parameter );
    public Parameter Add( string name, object value );

```



```

    public bool Contains( Parameter parameter );
    public bool Contains( string name );

    public void CopyTo( Parameter[] array, int index );
    void ICollection.CopyTo( Array array, int index );

    IEnumerator IEnumerable.GetEnumerator();

    public void Remove( Parameter parameter );
    public void Remove( string name );
}

// Represents a search that has been optimized for repeated execution.
public class PreparedFind
{
    public ItemContext ItemContext {get;}

    public ParameterCollection Parameters {get;}

    public FindResult FindAll();

    public object FindOne();

    public object FindOnly();

    public bool Exists();
}

// Specifies sorting options used in a search.
public class SortOption
{
    // Initialize a object with default values.
    public SortOption();

    // Initializes a SortOptions object with SearchExpression, order.
    public SortOption( SearchExpression searchExpression, SortOrder order );

    // A search SearchExpression that identifies the property that will be sorted.
    public SearchExpression Expression {get; set;}

    // Specifies ascending or descending sort order.
    public SortOrder Order {get; set;}
}

// A collection of sort option objects.
public class SortOptionCollection : IList
{
    public SortOptionCollection();
}

```

```

public SortOption this[int Index] {get; set;}

public int Add( SortOption value );
public int Add( StringExpression expression, SortOrder order );
int IList.Add( object value );

public void Clear();

public bool Contains( SortOption value );
bool IList.Contains( object value );

public void CopyTo( SortOption[] array, int index );
void ICollection.CopyTo( Array array, int index );

public int Count {get;}

IEnumerator IEnumerable.GetEnumerator();

public void Insert( int Index, SortOption value );
void IList.Insert( int Index, object value );

public int IndexOf( SortOption value );
int IList.IndexOf( object value );

public void Remove( SortOption value );
void IList.Remove( object value );
public void RemoveAt( int index );

public object SyncRoot {get;}
}

// Specifies the sort order using in a SortOption object.
public enum SortOrder
{
    Ascending,
    Descending
}
}

```

## APPENDIX C

```

namespace System.St rage
{

    public abstract class FindResult : IAsyncObjectReader
    {

        public FindResult();

        // Moves the FindResult to the next position in the result.
        public bool Read();
        public IAsyncResult BeginRead( AsyncCallback callback, object state );
        public bool EndRead( IAsyncResult asyncResult );

        // The current object.
        public object Current {get;}

        // Returns whether or not the FindResult contains any objects.
        public bool HasResults {get;}

        // Returns whether or not the FindResult is closed.
        public bool IsClosed {get;}

        // Returns the type of items in this FindResult.
        public Type ObjectType {get;}

        // Closes the FindResult
        public void Close();
        void IDisposable.Dispose();

        // Returns an enumerator over the FindResult, starting at the current position. Advancing
        // any enumerator on the FindResult advances all enumerators as well as the FindResult
        // itself.
        IEnumerator IEnumerable.GetEnumerator();
        public FindResultEnumerator GetEnumerator();

    }

    public abstract class FindResultEnumerator : IEnumerator, IDisposable
    {

        public abstract object Current { get; }
        public abstract bool MoveNext();
        public abstract void Reset();
        public abstract void Close();

        void IDisposable.Dispose();

    }

}

namespace System

```

```

{
    // A common interface for iterating over objects.
    public interface IObjectRead r : IEnumerable, IDisposable
    {
        object Current {get;}
        bool IsClosed {get;}
        bool HasResults {get;}
        Type ObjectType {get;}

        bool Read();
        void Close();
    }

    // Adds asynchronous methods to IObjectReader
    public interface IAsyncObjectReader : IObjectReader
    {
        IAsyncResult BeginRead( AsyncCallback callback, object state );
        bool EndRead( IAsyncResult result );
    }
}

```

*[Remainder of Page Intentionally Left Blank]*